A Transformational Approach to High Performance Embedded Computing

Wim Böhm
Colorado State University
Fort Collins, CO

Jeffrey Hammes
SRC Computers, Inc.
Colorado Springs, CO

1. Introduction

This paper describes a transformational, high level language approach to High Performance Embedded Computing on the SRC-6 machine and its MAP^{TM} reconfigurable hardware. A program is initially written in pure C and compiled by the MAP C Compiler. Then, using feedback from the MAP C compiler, the program is successively transformed manually to achieve better performance. These transformations avoid certain inefficiencies, such as re-reading values from memory, loop slowdown caused by loop carried dependencies, and underutilizing memory bandwidth. We discuss the transformations in the context of the Wavelet Versatility Benchmark and the Gauss-Seidel iterative linear equation solver.

FPGAs use a large number of pins to connect to memories. They do not have caches, but they have on-chip block RAM, allowing the programmer to decide what data stays on chip. Also, fine grain operation level parallelism combined with pipelining makes it possible for FPGAs to execute an inner loop body in one clock cycle. These characteristics provide a simple, deterministic performance model, allowing the programmer to work towards a well defined goal: store hot data structures on chip either in block RAM or in registers, create inner loop bodies that execute in one clock cycle and use the full memory bandwidth of the machine by loop unrolling.

2 The SRC-6 and MAP Compiler

The SRC-6 machine contains a pair of dual-processor Pentium IV boards, running Linux, and two SRC-developed FPGA-based reconfigurable processors called MAPs. Each MAP contains two Xilinx Virtex$-II^{TM}$ FPGAs, six banks of dual-ported SRAM On Board Memory (OBM) totaling 24 Mbytes, and a control FPGA containing a DMA engine that manages memory transfers into and out of the OBM. DMAs can take place concurrently with executing FPGA code. Both user FPGAs have access to the OBM banks, though only one can access a given memory at a time. Each of the six memory banks contains 512K 64-bit words (4 Mbytes). The user FPGAs are clocked at a fixed frequency of 100 MHz. Each OBM bank can handle one write or read from a user FPGA in each clock.

Using a simple directive the user allocates off chip arrays onto OBM banks. The MAP Compiler allocates local arrays to the FPGAs block RAM and local scalar variables in registers. The MAP compiler front end produces a control flow graph (CFG) of basic blocks and directed control flow edges between the blocks. Next the MAP Compiler translates each block into its own dataflow graph (DFG) that exposes instruction-level parallelism. It then merges the DFG fragments that compose an innermost loop into a single pipelined code block that includes a driver module for firing loop iterations. The driver will fire one loop iteration on each clock, unless data dependencies or multiple accesses to a bank force it to run slower. The DFGs for the code blocks are then mapped to Verilog, using straightforward instantiations of pre-defined macros. The Verilog is synthesized and place-and-routed using commercial software.

The MAP Compiler allows a user to create "user-macros" in Verilog. Their semantics differs from functions in C in that they can retain state between calls. This allows for program transformations providing code optimization beyond straight forward C compilation.

3 Wavelet Versatility Benchmark

The Wavelet Versatility Benchmark is part of a benchmark suite for evaluating configurable computing systems. This suite was produced by Honeywell as part of the DARPA/ITO ACS (Adaptive Computing Systems) program [2]. The Wavelet Benchmark consists of four phases: Wavelet Transform, Quantization, Run-Length Encoding and Entropy Encoding.

The Wavelet Transforms perform a 5x5 convolution of an input image stepping by 2 in both horizontal and vertical directions, reading from one OBM and writing to four. In the initial pure C implementation of this convolution, values are re-read on average 2.5 times. This can be avoided by using system-macros that implement a delay queue mechanism [1]. In the next transformation, four pixels are packed in one word, and two horizontally adjacent convolutions are

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 FEB 2005	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Transformational Approach to High Performance Embedded Computing	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Colorado State University Fort Collins, CO**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM00001742, HPEC-7 Volume 1, Proceedings of the Eighth Annual High Performance Embedded Computing (HPEC) Workshops, 28-30 September 2004 Volume 1., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**UU**	**41**	

performed in parallel.

The initial implementation of the Quantization, Run-Length Encoding and Entropy Encoding phases read/write pixels (packed after the Wavelet phase was optimized) from/to OBMs. To avoid unnecessary memory traffic and to fully benefit from pipelining, the three phases are loop fused. The MAP compiler indicates loop slowdown, caused by loop carried data dependencies in Run-Length Encoding and Entropy Encoding in the form of (min) reductions and (summing and shifting) accumulations. These can be avoided by using stateful reduction and accumulator macros.

A final transformation distributes the program in a coarse grain parallel fashion over the two user FPGAS. Its execution on the MAP produces bit-identical results to the Honeywell reference code and achieves a speedup of 38 when compared to the reference code executed on a 2.8 GHz Pentium IV.

4 Gauss Seidel

Gauss Seidel is an iterative linear system solver of diagonally dominant systems $Ax = b$. The inner loop of the code recomputes $x[i]$ using a vector inproduct:

```
for(i=0;i<n;i++) {
 s = 0.0;
 for(j=0;j<n;j++)
  if (j != i)  s += A[i*COL+j] * x[j];
 x[i] = b[i] - s;
}
```

All A, b and x values are in single precision floating point. In a first pure C implementation the x vector is allocated in block RAM, while A and b are allocated in one OBM. When this code is compiled, the MAP compiler indicates a loop slowdown because s is both read and written in the inner loop. This can be avoided by using a floating point accumulator macro. This accumulator needs to be able to accept a new input every clock cycle. It will have a larger than one latency, as a floating point add takes 10 cycles on the MAP. This implies that the accumulator will have to be parallelized internally. At the time of writing this abstract, this and other floating point macros have not been integrated in the MAP compiler yet.

In a next program transformation, k values $x[i], x[i + n/k], ..x[i + (k - 1)n/k]$ are updated in the inner j loop in parallel. Because we can row block partition A and b over six OBMs, a good value for k is 6. The j loop reads one value from each OBM and performs 6 multiplies and 6 adds.

Because we are using single precision floating point we can pack two adjacent values in one word, thereby doubling the amount of computation per communication. The inner loop now performs 12 multiplies and 12 adds in each iteration. However, the MAP compiler now reports a loop slowdown: because the inner loop is unrolled, two consecutive x values are read from block RAM. This can be avoided by stripe partitioning the odd and even x elements over two block RAMs.

The code currently runs in debug mode on a host machine. The MAP compiler backend for floating point operations will soon become available, and we will assess the hardware performance of the Gauss Seidel codes. If the most parallel version of the code with the 12 single clock floating point accumulators can be placed and routed on the FPGA, its inner loop will execute 24 floating point operations per clock cycle. At 100 MHz, this will represent 2.4 GFlops.

5 Conclusions and Future Work

In this paper we have argued that high performance embedded computing can be achieved on the SRC-6 machine and its MAP reconfigurable hardware by starting from a pure C code and transforming this code stepwise using system or user macros. For the codes we have studied, the transformations 1) employ delay queues to avoid re-reading from OBMs, 2) pack data items in words and 3) unroll loops to increase bandwidth, 4) use accumulator macros to avoid loop slowdown caused by loop carried dependencies, 5) fuse loops to avoid memory traffic, 6) partition arrays to avoid multiple memory accesses in the same loop body, and 7) perform coarse grain task parallelization to shorten the critical path of the complete application. In future work we will implement sparse solvers and the NAS Parallel Benchmark suite.

References

[1] J. Hammes. Methodology for pipelining and fusing stenciled loops. Technical Report SWP-009-00, SRC Computers, Inc., November 2003.

[2] R. Kohler. Benchmark specification document – versatility stressmark. Technical Report CDRL A001, Rome Laboratory, November 1997. Submitted by Honeywell, Inc.

A Program Transformation Approach to High Performance Embedded Computing using the SRC MAP® Compiler

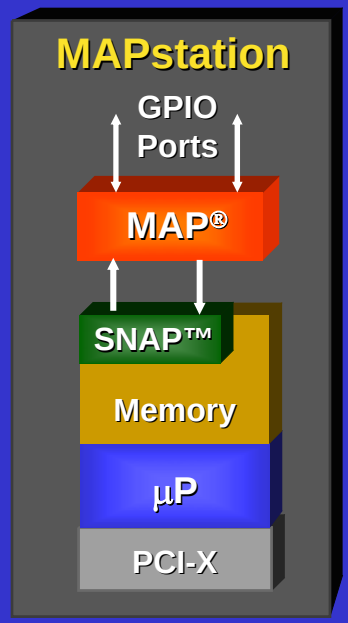
Wim Bohm, Colorado State University

and

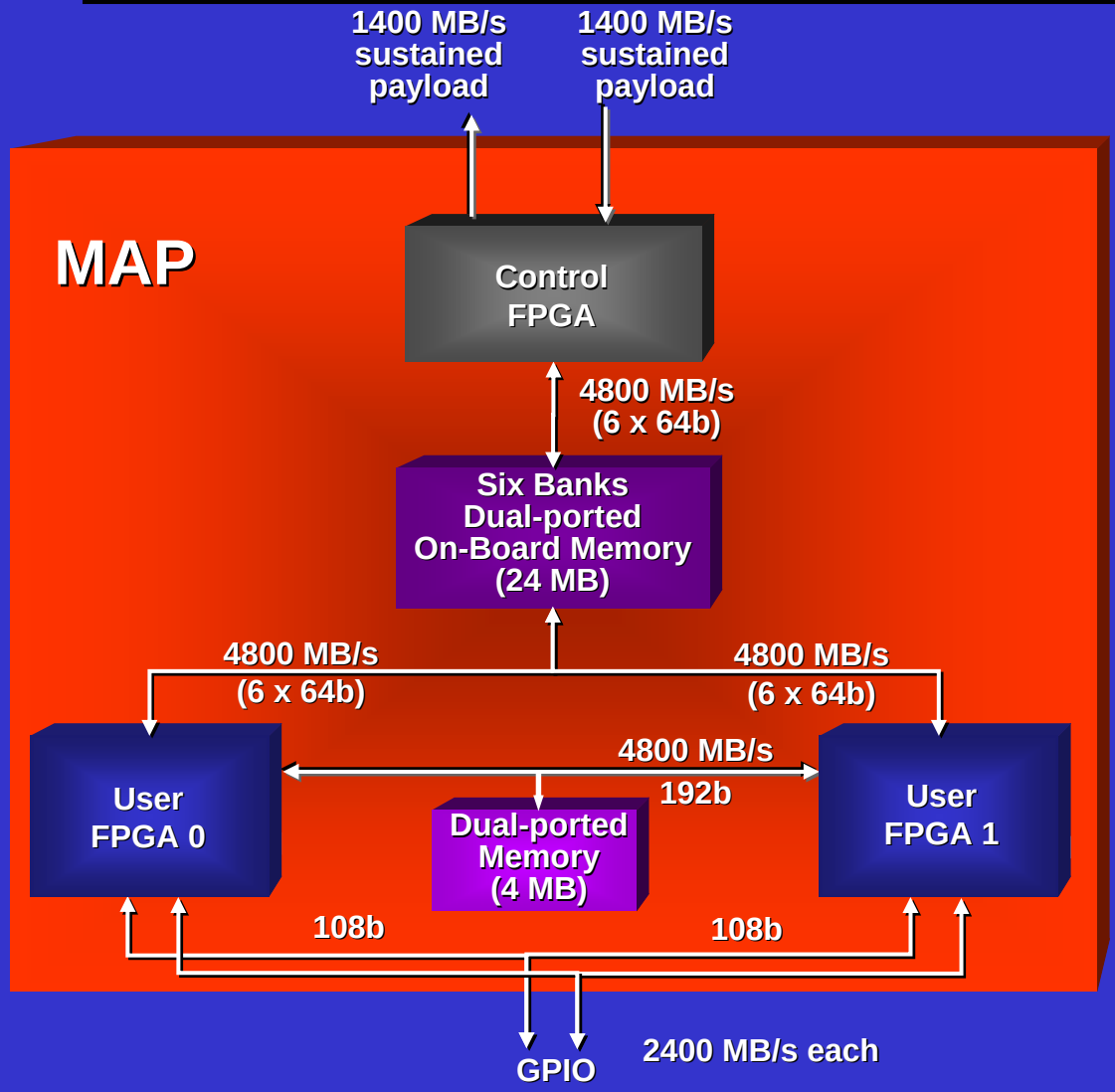
Jeff Hammes, SRC Computers, Inc.

SRC–6 MAP® System

- SRC-6 MAP
  - FPGA based High Performance architecture
  - Fortran / C compiler for the whole system
- One Node:
  - Microprocessor
  - MAP reconfigurable hardware board
  - SNAP µproc and MAP interconnected via DIM slot
  - GPIO ports allow connection to other MAPs
  - PCI-X can connect to other µprocs
- Multiple configurations / implementations
  - this talk: MAPstation - one node
- MAP C Compiler
  - Compiler generates both µproc and MAP code
  - user partitions µproc, MAP tasks

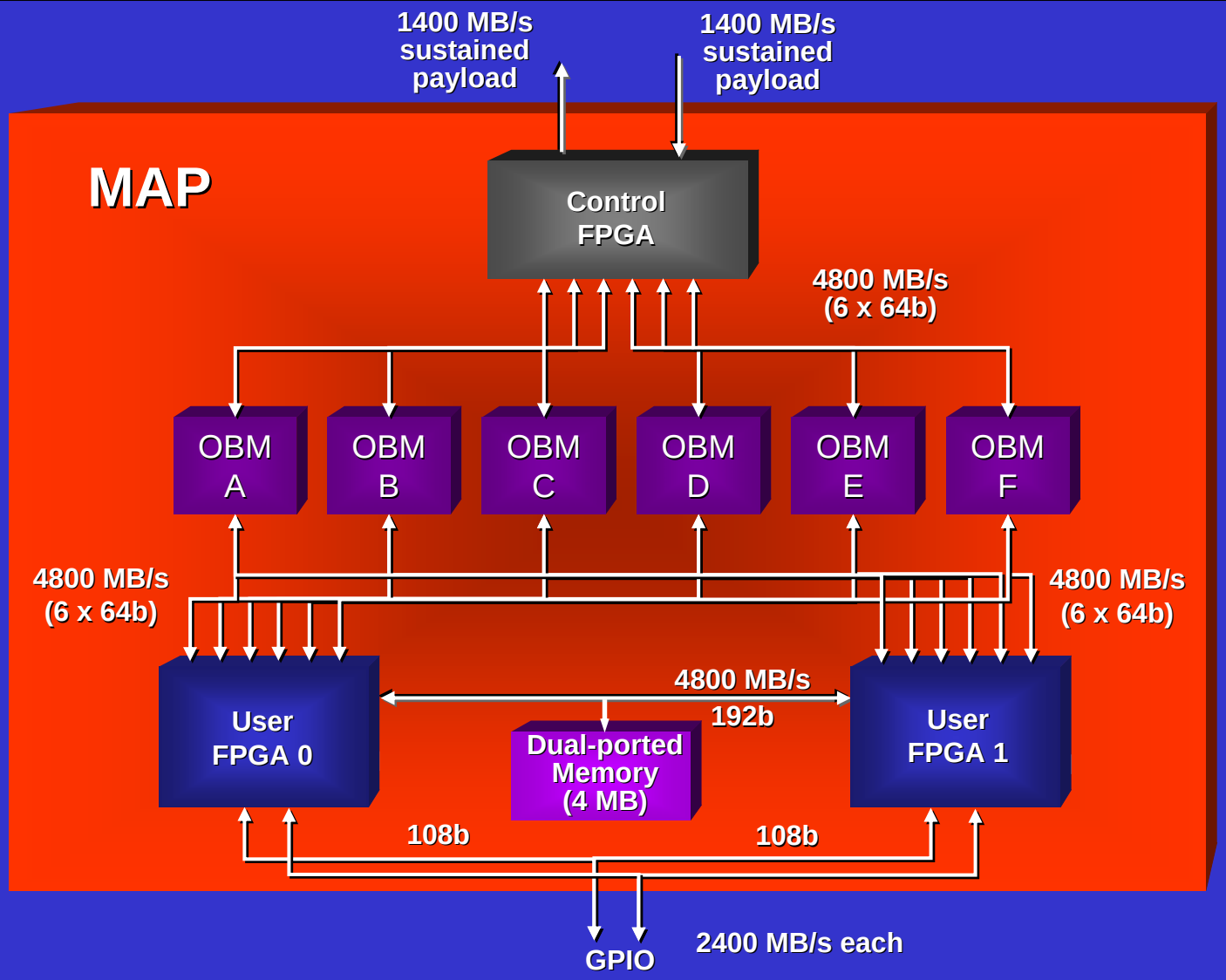


MAP® board architecture



- Direct Execution Logic (DEL) made up of one or more User FPGAs
- Control FPGA performs off board memory access
- Multiple banks of On-Board Memory maximize local memory bandwidth
- GPIO ports allow direct MAP to MAP chain connections or direct data input
- Multiple parallel data transports:
  - Distributed SRAM in FPGA
    - 264 KB @ 844 GB/s
  - Block SRAM in FPGA
    - 648 KB @ 260 GB/s
  - On-Board SRAM Memories
    - 28 MB @ 9.6 GB/s
  - Microprocessor Memory
    - 8 GB @ 1400 MB/s

MAP Programmers View



MAP C compiler

- **Pure C runs on the MAP !!**
- **MAP C Compiler**
  - **Intermediate form: dataflow graph of basic blocks**
  - **Generated code: circuits**
    - **Basic blocks in outer loops become special purpose hardware “function units”**
    - **Basic blocks in inner loop bodies are merged and become pipelined circuits**
- **Sequential semantics obeyed**
  - **One basic block executed at the time**
  - **Pipelined inner loops are slowed down to disambiguate read/write conflicts if necessary**
  - **MAP C compiler identifies (cause of) loop slowdown**

Execution Modes

- **DEBUG Mode**
  - code runs on workstation
  - allows debugging (printf ☺)
  - allows most performance tuning (avoiding loop slow downs)
  - user spends most time here
- **Two SIMULATION Modes**
  - Dataflow level and Hardware level
  - mostly used by compiler / hardware function unit developers
  - very fine grain information
- **HARDWARE Mode**
  - final stage of code development
  - allows performance tuning using timer calls

Transformational Approach

- Start with pure C code
- Partition Code and Data
  - distribute data over OBMs and Block RAMs
  - distribute code over two FPGAs
    - only one chip at the time can access a particular OBM
    - MPI type communication over the bridge
- Performance tune (removing inefficiencies)
  - avoid re-reading of data from OBMs using Delay Queues
  - avoid read / write conflicts in same iteration
  - avoid multiple accesses to a memory in one iteration
  - avoid OBM traffic by fusing loops
- Today's transformation is tomorrow's compiler optimization

How to performance tune: Macros

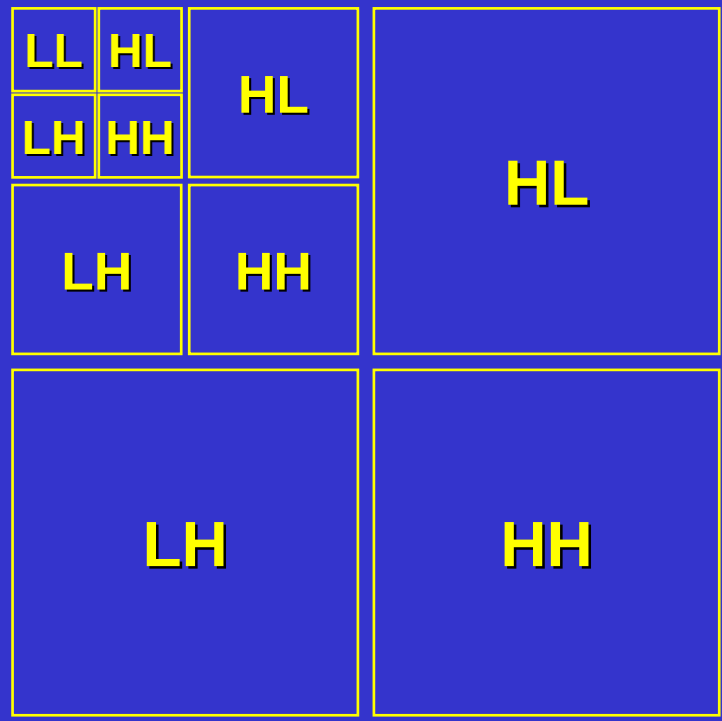
- C code can be extended using macros allowing for program transformations that cannot be expressed straightforwardly in C
- Macros have semantics unlike C functions
  - have a period (#clocks between inputs)
  - have a pipeline delay (#clocks between in and output)
  - MAP C compiler takes care of period and delay
  - can have state (kept between macro calls)
  - two types of macros
    - system provided
      - compiler knows their period and delay
    - user provided (written in e.g. Verilog)
      - user needs to provide period and delay

Two Case Studies

- **Wavelet Versatility Benchmark**
  - Image processing application (wavelet compression)
  - Part of DARPA/ITO ACS (Adaptive Computing Systems) benchmark suite
  - **Versatile**: Four phases of **different computational nature**
    - 1: wavelet transform: *window access, multiple outputs*
    - 2: quantization: *sum, min, max reductions*
    - 3: run length encoding: *while loop, irregular output*
    - 4: Huffman encoding: *table lookups*
- **Gauss Seidel Linear Equation Solver**
  - Numerical (Floating Point) kernel
  - Iterative nature: *non-perfect loop structure*
  - Many applications

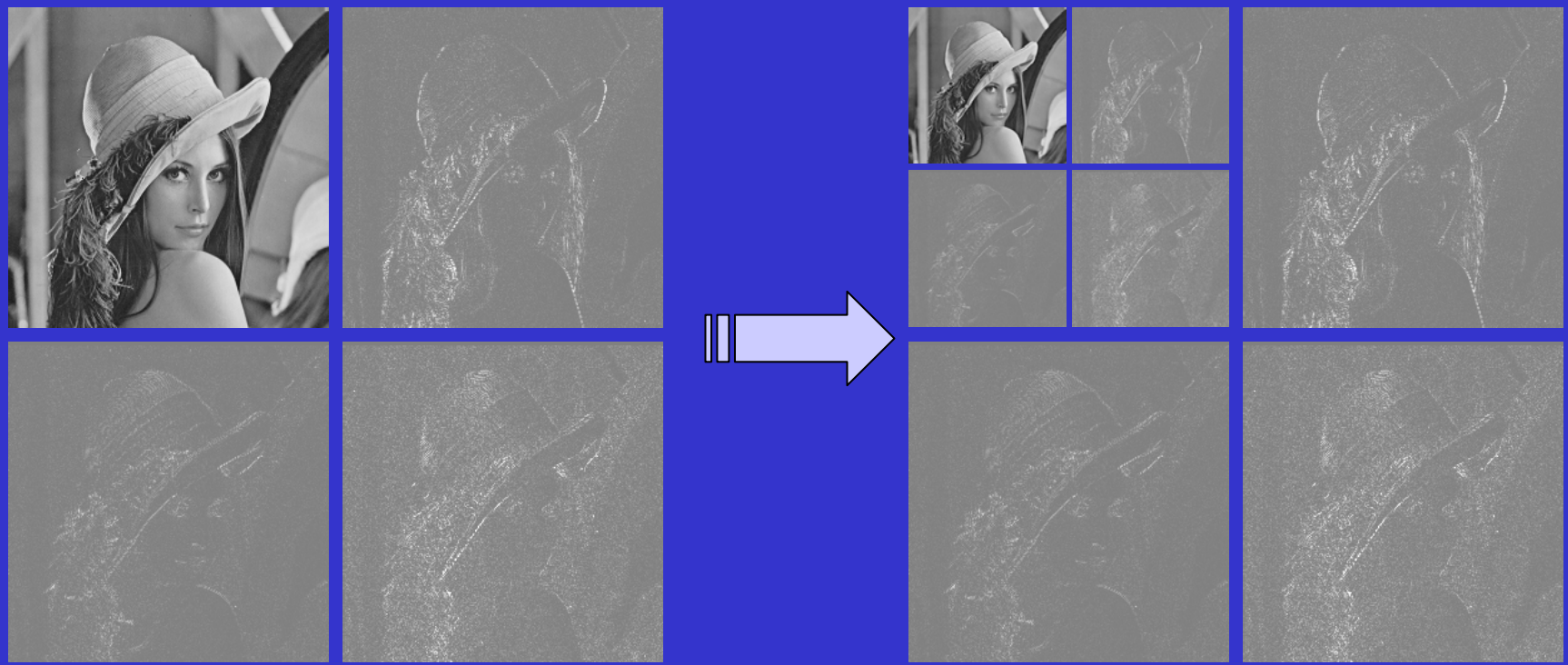
Wavelet Versatility Benchmark

- **Wavelet transform**
  - Applied three times
    - Second and third passes use upper left quadrant of previous pass
  - L: Low pass filter (average)
  - H: High pass filter (derivative)
- **Wavelet does not compress but enables compression in further stages (many 0s in H)**
  - Quantization
  - Run-Length Encoding
  - Huffman Encoding

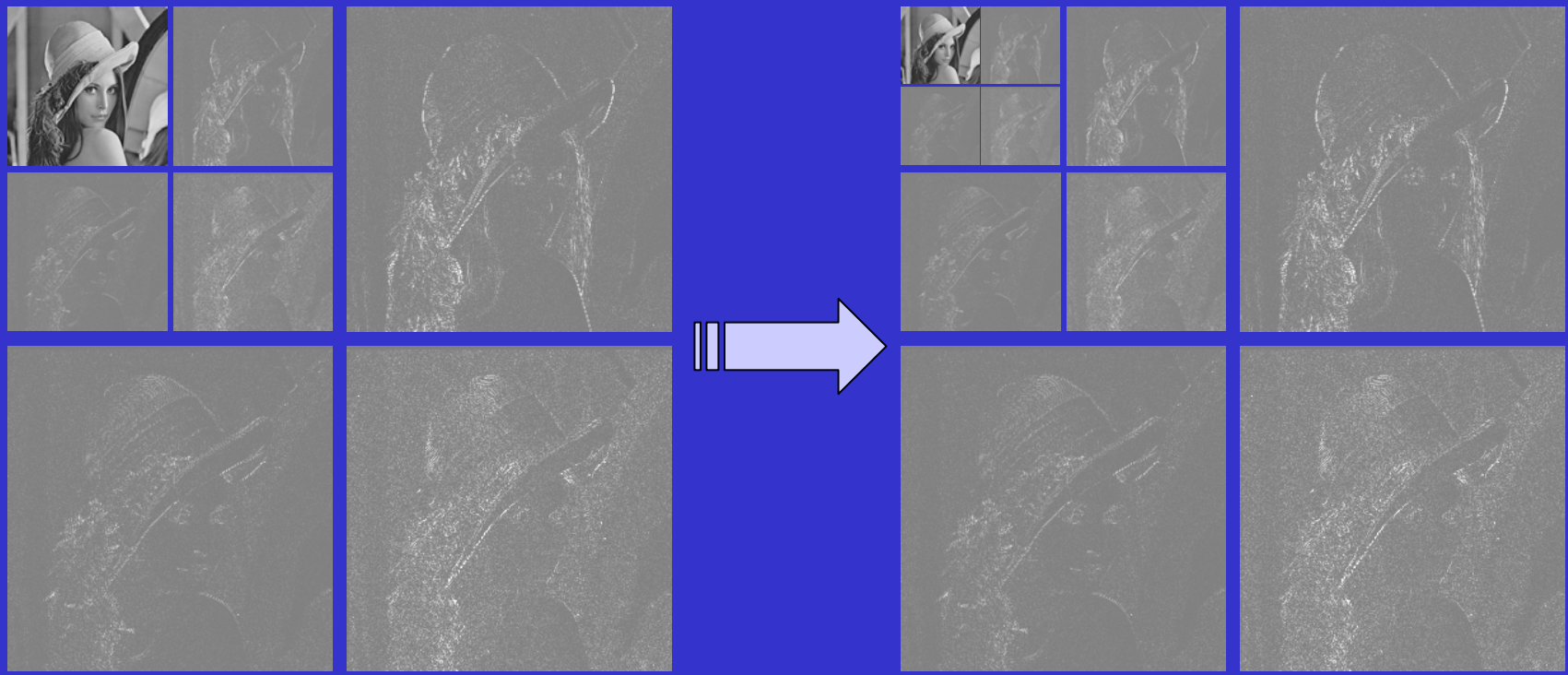


First wavelet step

Second wavelet step



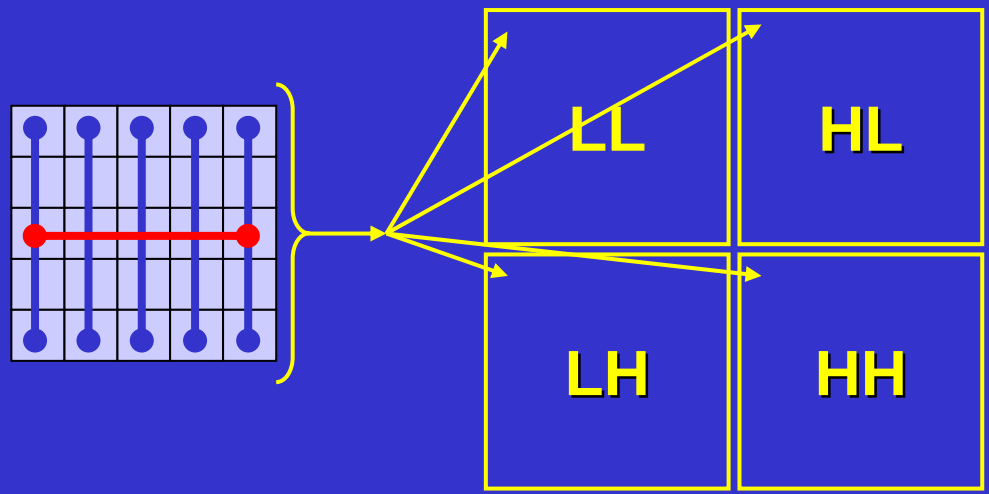
Final wavelet step



MAP C Algorithm

One 5x5 window stepping by 2 in both directions

- Computes LL, LH, HL, and HH simultaneously

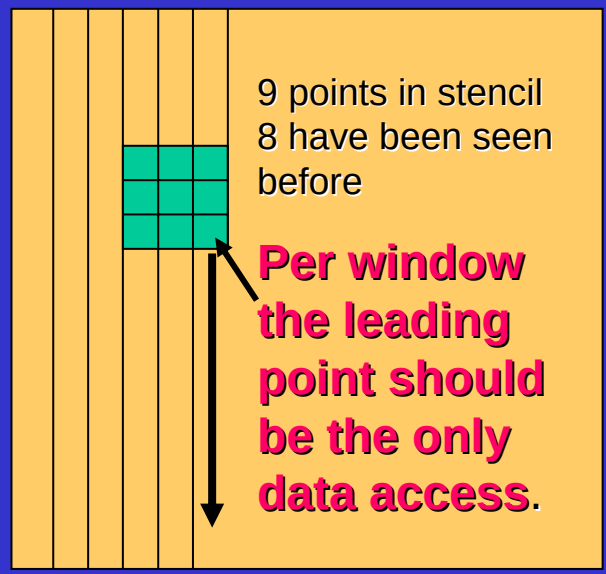


- **Inefficiency**: naive first implementation re-accesses overlapping image elements

Efficient Window Access

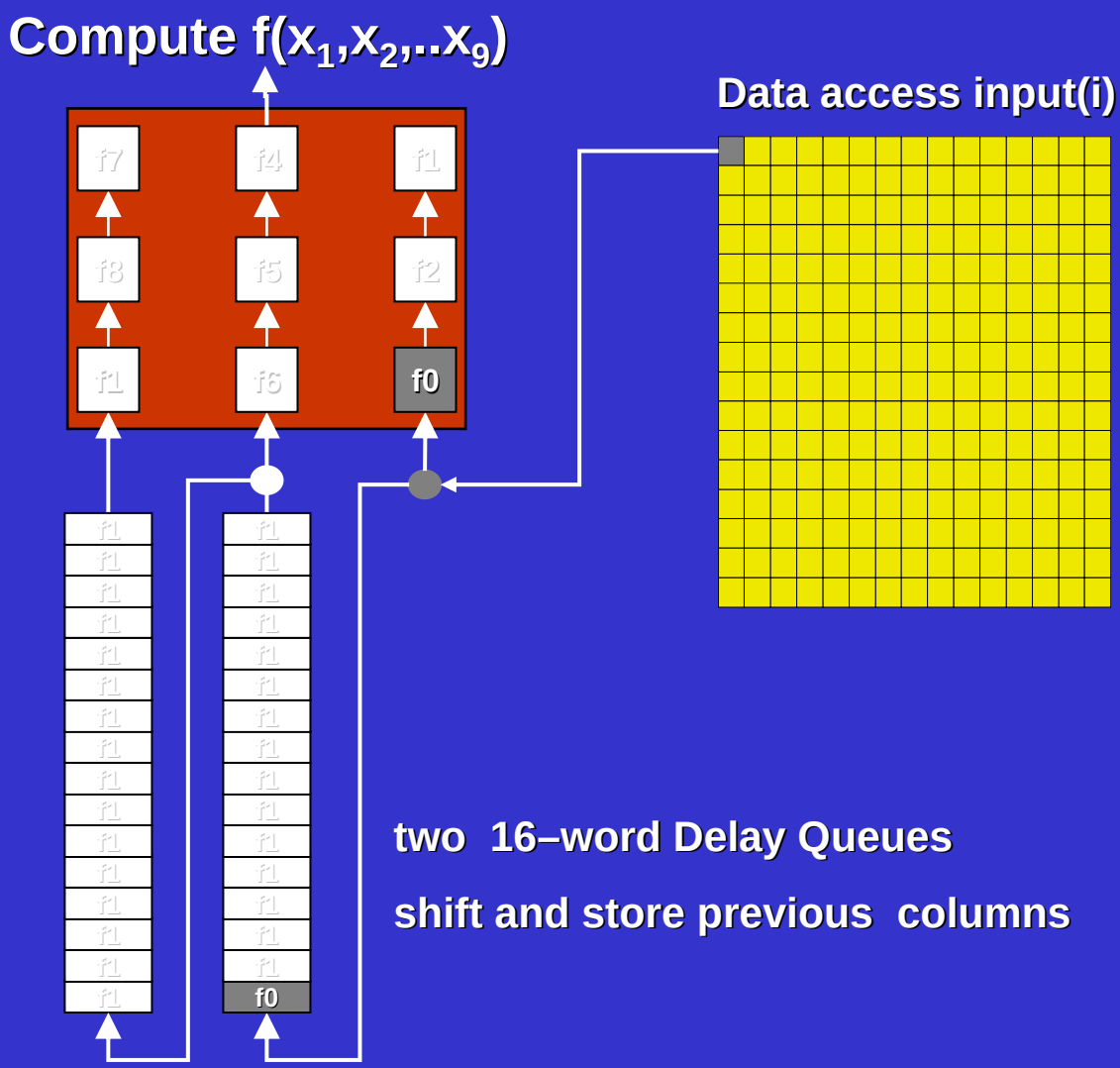
- **Keep data on chip using Delay Queues**
  - E.g. 16 deep (using efficient hardware SLR16 shifters)

- **Simplified example:**
  - 3x3 window
    - stepping 1 by 1
    - in column major order
  - image 16 deep
    - general case divides the Image in 16 deep strips

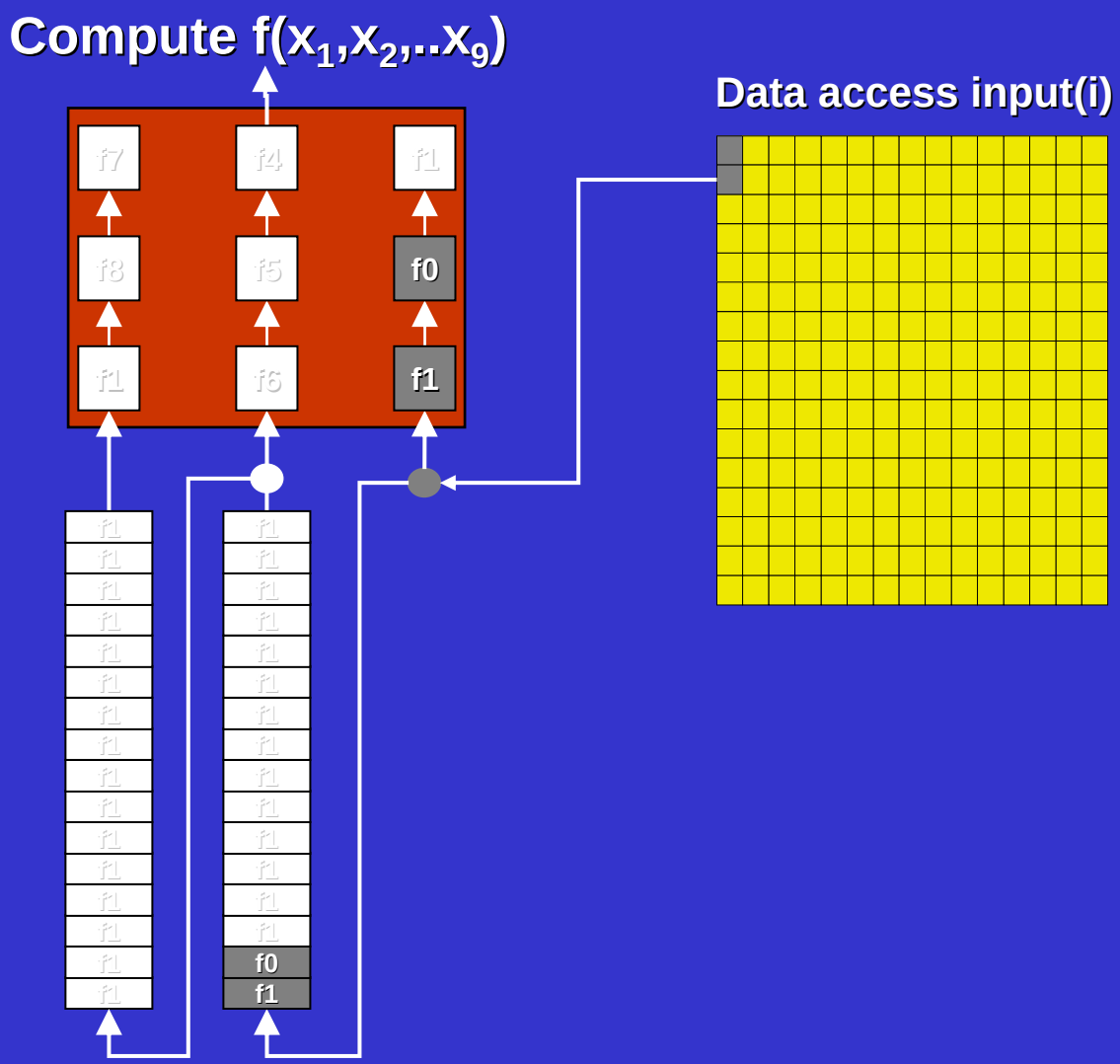


input array traversal

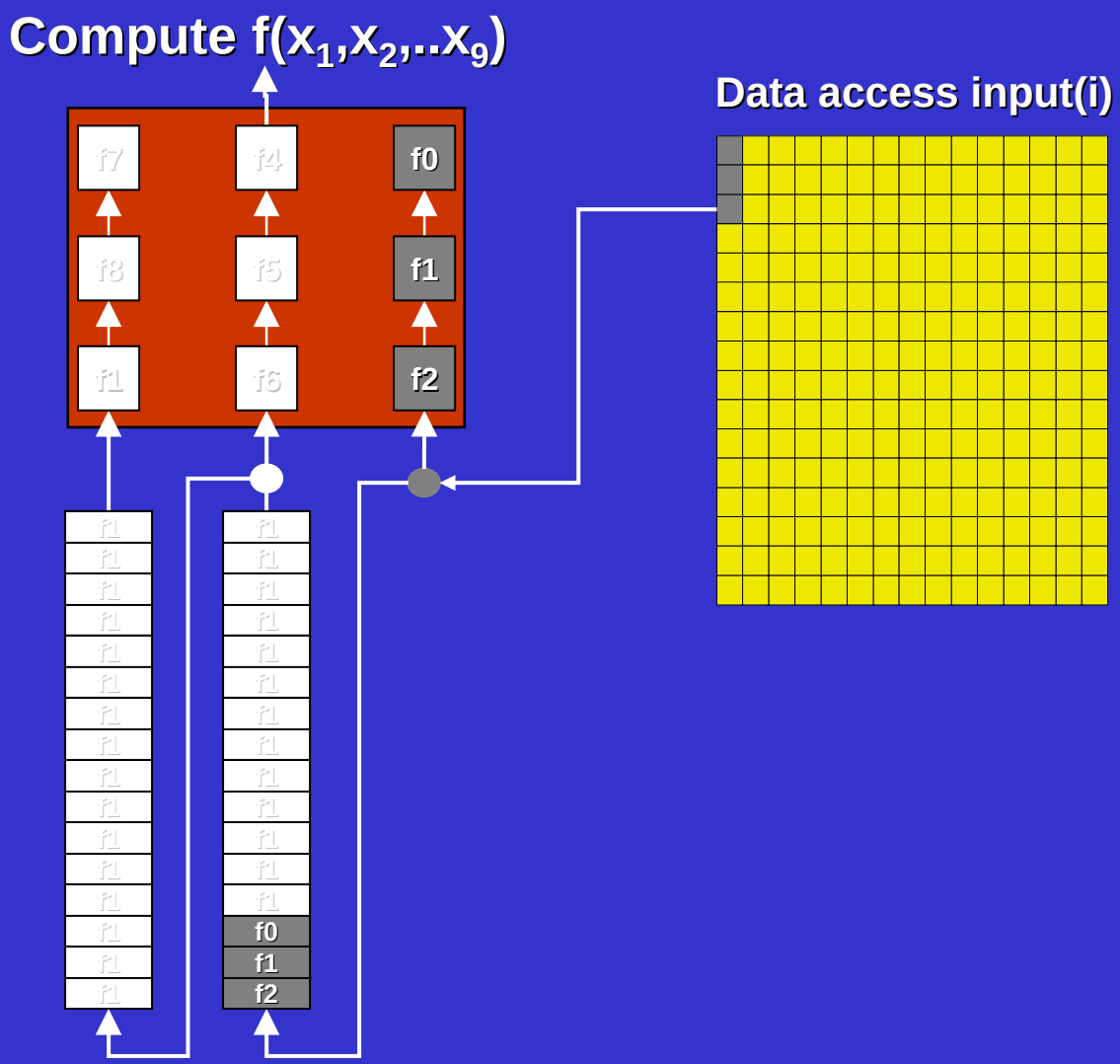
Delay Queues 1



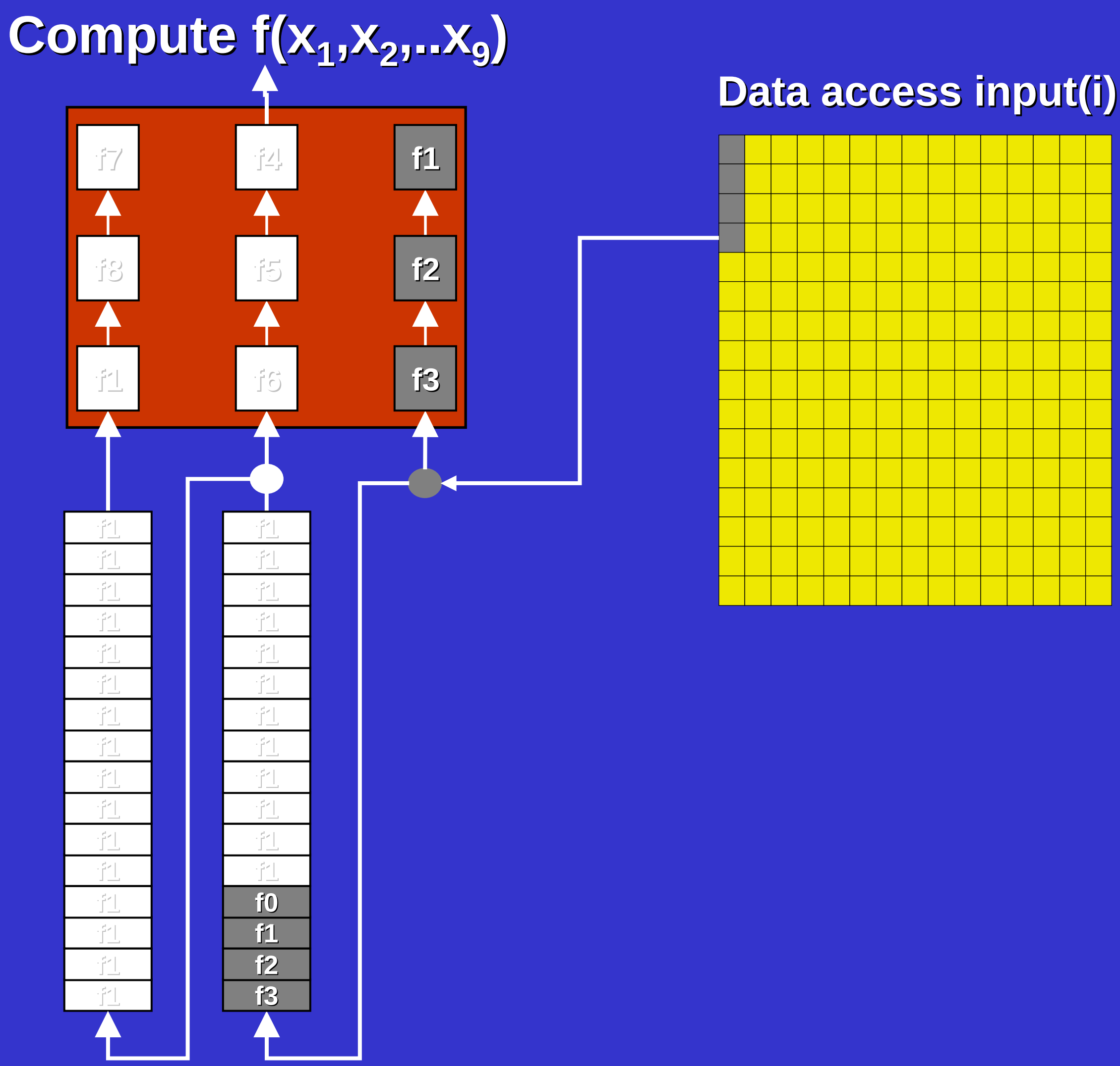
Delay Queues 2



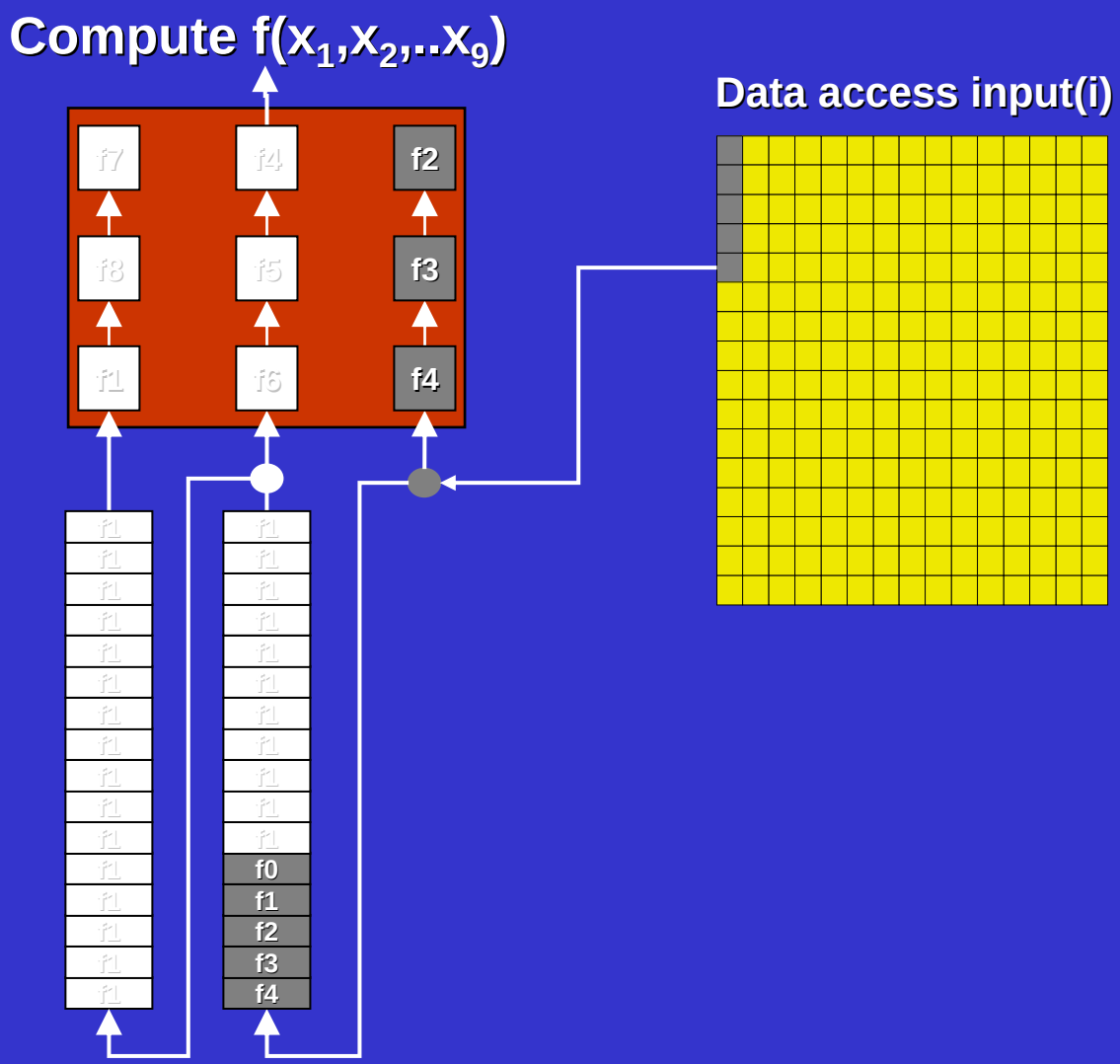
Delay Queues 3



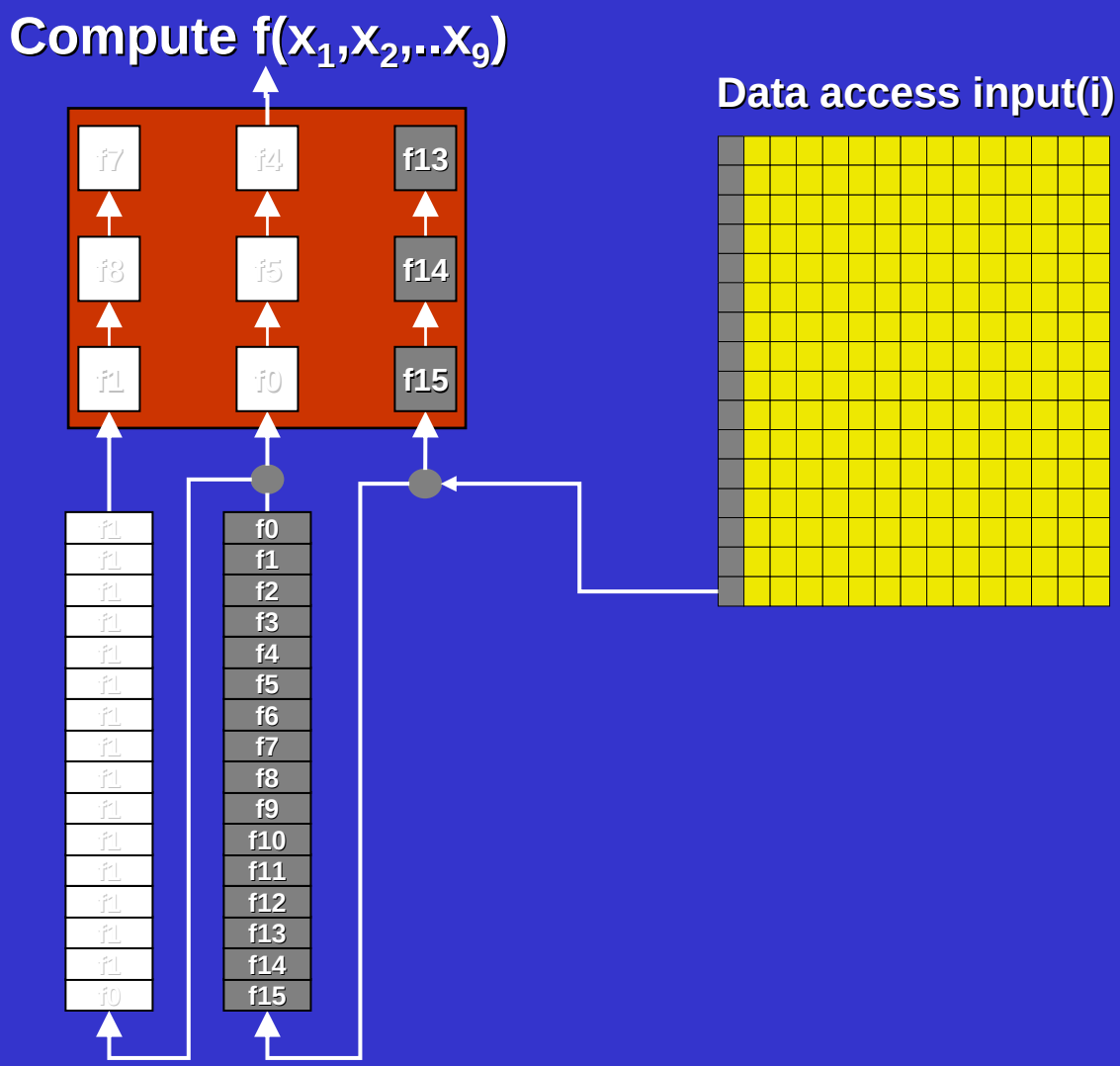
Delay Queues 4



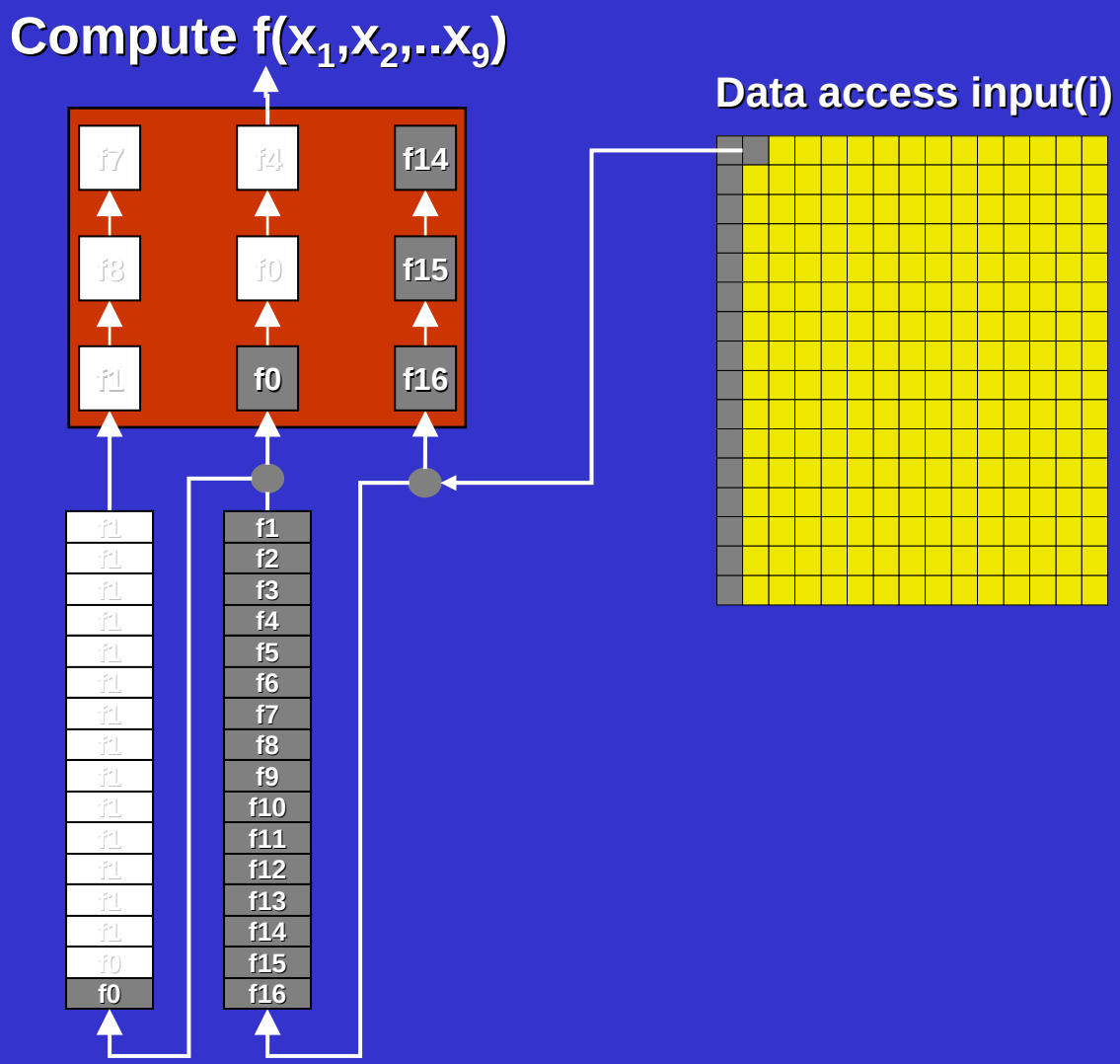
Delay Queues 5



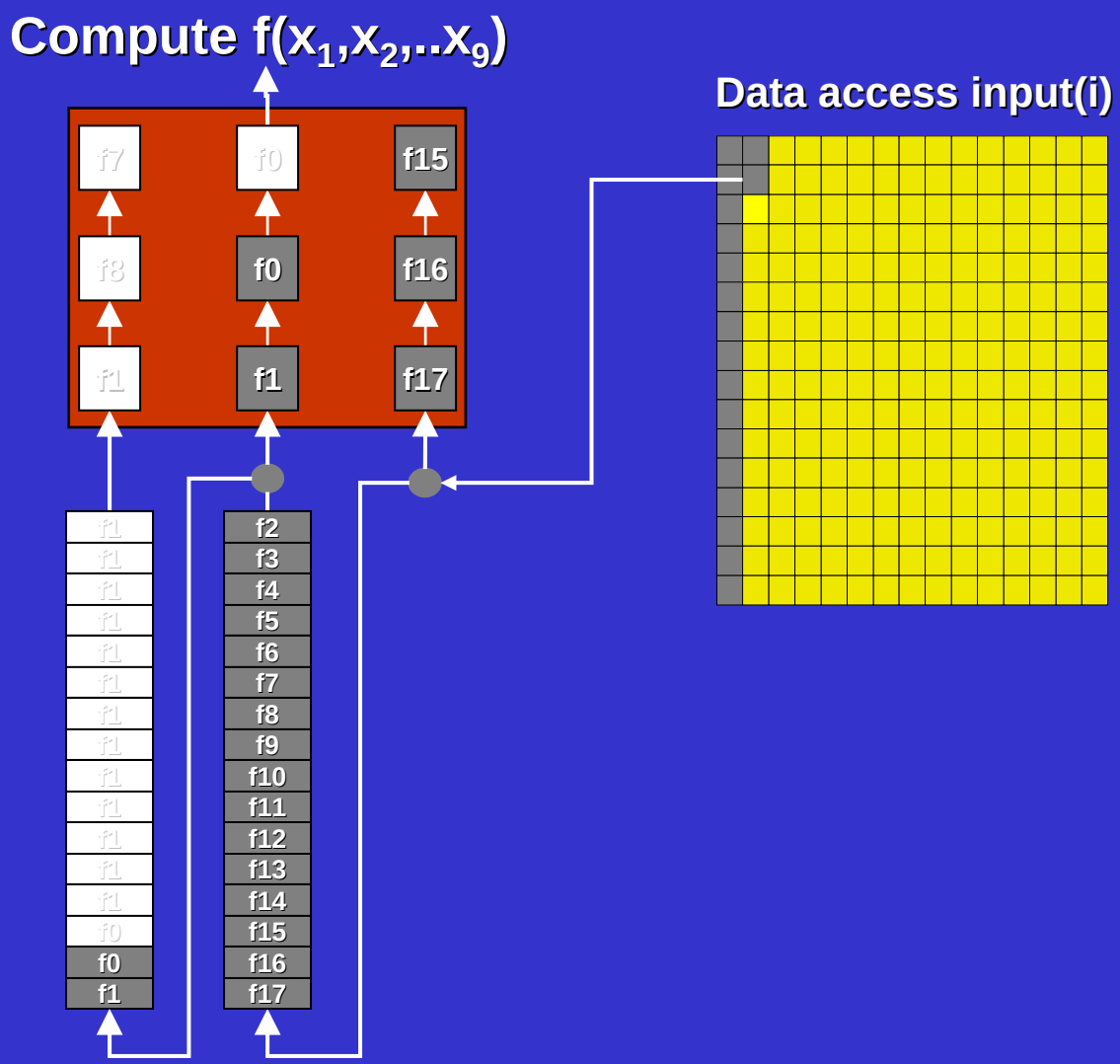
… Delay Queues 16



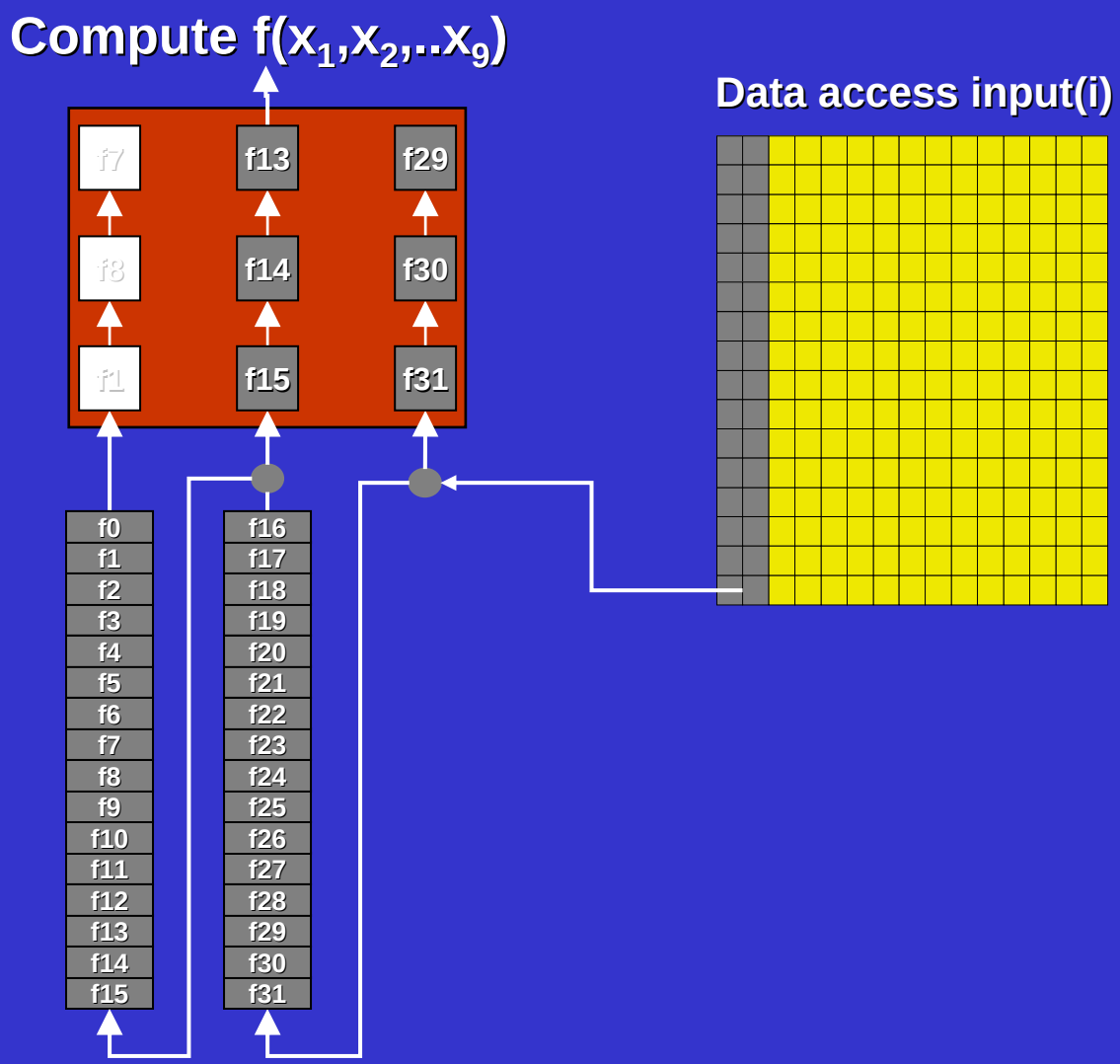
Delay Queues 17



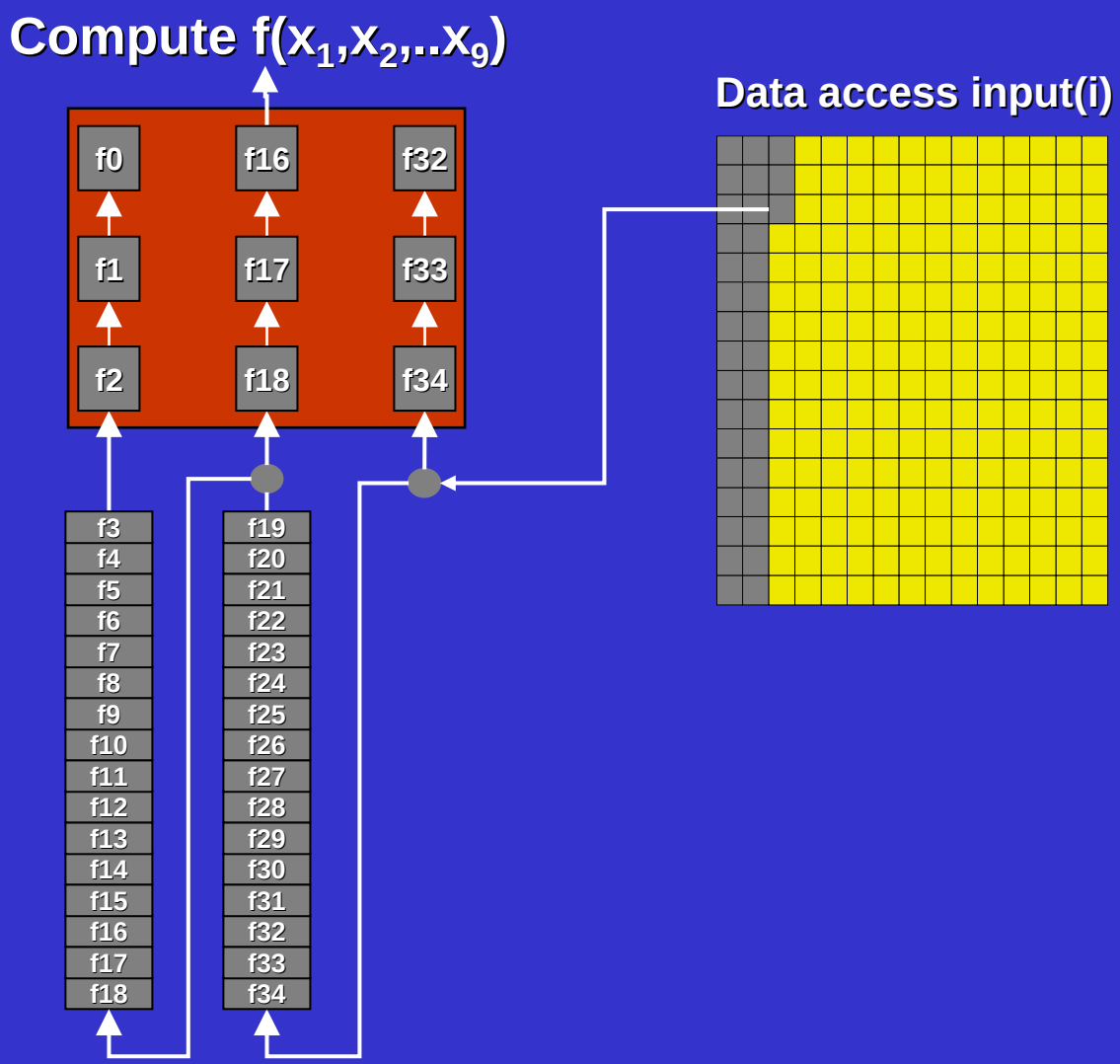
Delay Queues 18



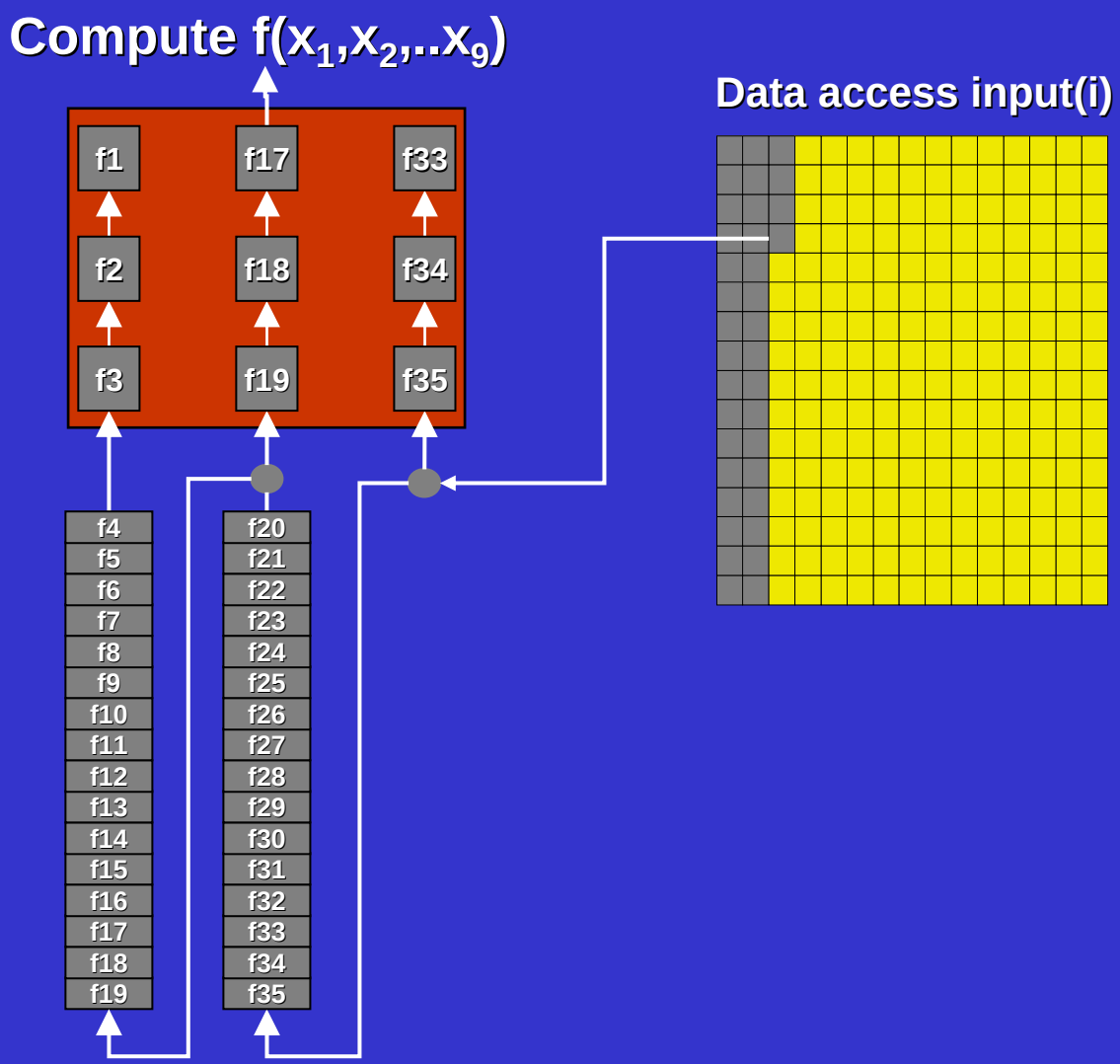
…Delay Queues 32



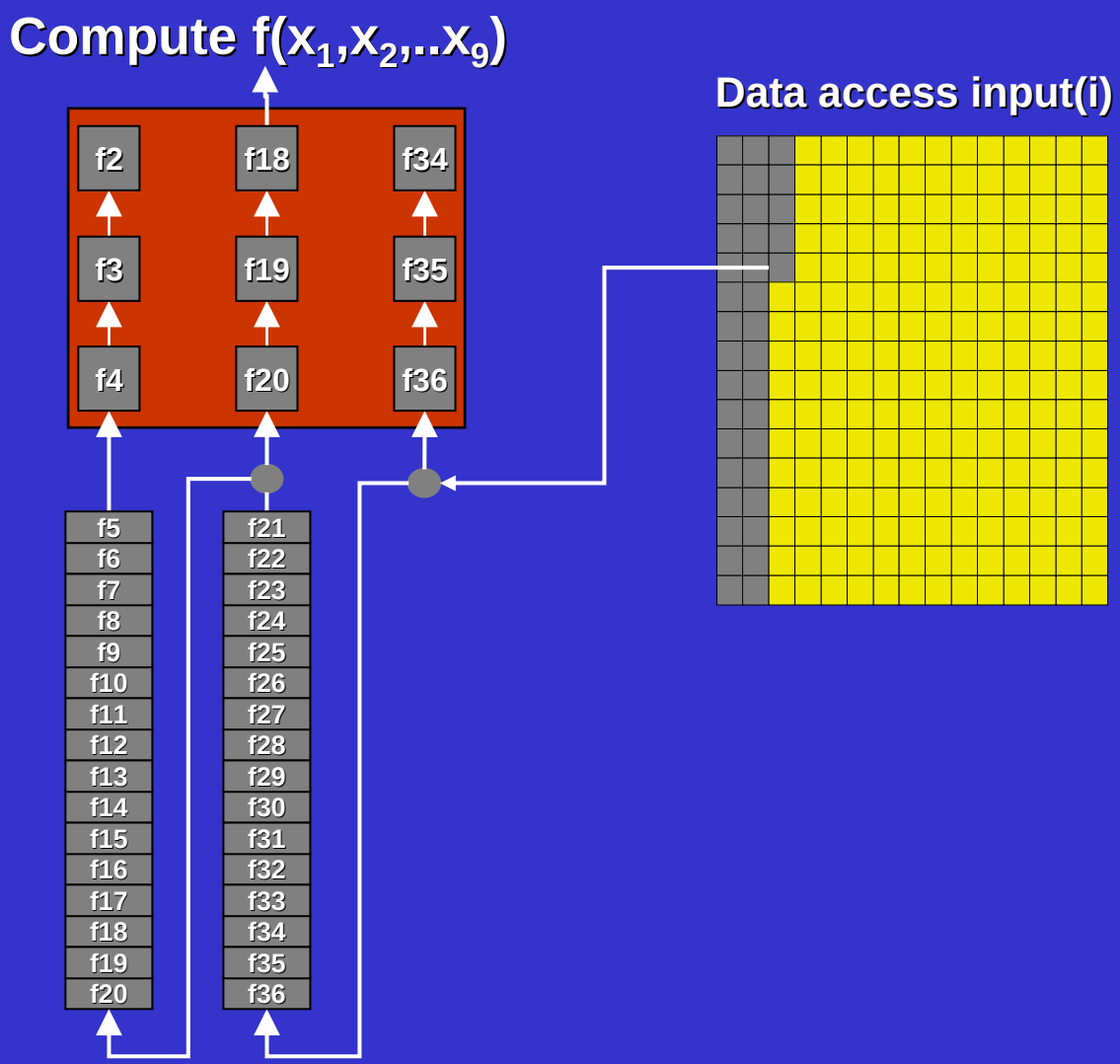
…Delay Queues 35



Delay Queues 36



Delay Queues 37



Delay Queues: Performance

3x3 window access code

512 x 512 pixel image

Routine Style	Number of clocks	Comment
Straight Window	2,376,617	close to 9 clocks per iteration 2,340,900: the difference is pipeline prime effect
Delay Queue	279,999	close to1 clock per iteration 262144: theoretical limit
FPGA timing behavior is very predictable		

Wavelet Benchmark cont'

- **Rest of the code:**
  - Quantize each block in 16 bins per block
  - Run Length Encode zeroes
    - Occur frequently in derivative blocks
  - Huffman Encode
- **Three transformations**
  - Fuse the three loops avoiding OBM traffic
  - Use accumulator macros to avoid R / W conflicts
    - (see Gauss Seidel case study)
  - Task parallelize the complete code over two FPGAs

Versatility Benchmark: Performance

- 512x512 image
- Bit true results as compared to reference code
- Full implementation: All phases run on FPGAs

- Reference code compiled using Intel C compiler executed on 2.8 GHz Pentium IV: 76.0 milli-sec
- MAP execution time: 2.0 milli-sec
- MAP Speedup vs. Pentium 38

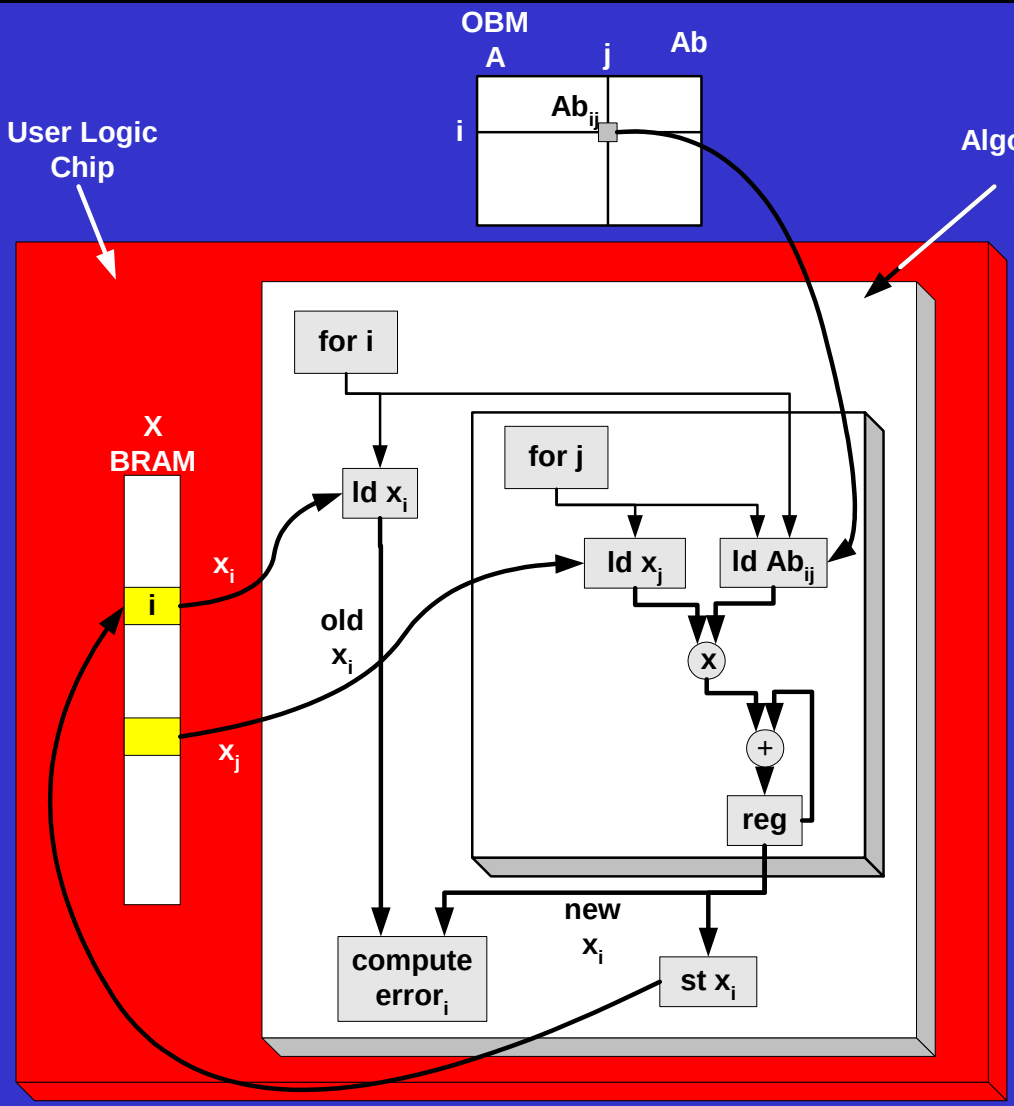
Gauss Seidel Iterative Solver

- Scientific Floating Point Kernel (single precision for now)
- Works for diagonally dominant matrices
- Some math manipulation to create an iterative solver:

$$Ax = b \rightarrow (L+D+U)x = b \rightarrow x = D^{-1}b - D^{-1}(L+U)x \rightarrow x_{n+1} = (Ab)x_n$$

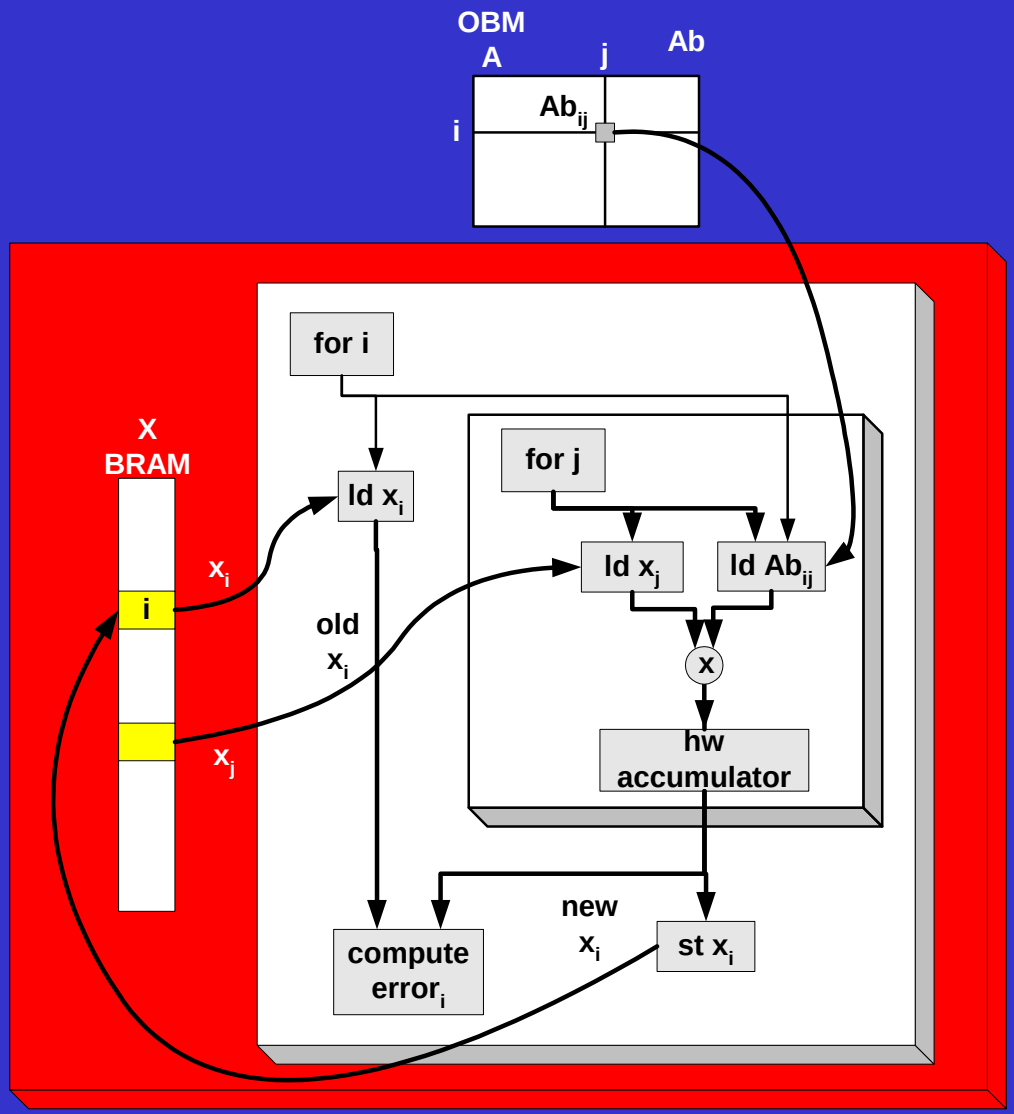
```
while(maxerror > tolerance) {       // do a next iteration
  maxerror = 0.0;
  for(i=0;i<n;i++) {                // compute new x[ i ]
    sxi = x[ i ];
    xi = 0.0;
    for(j=0;j<n+1;j++)
      xi += Ab[ i*COL+j ] * x[ j ];     // in product
    error = abs(xi – sxi);
  }
  maxerror = max(maxerror, error);
}
```

Pure C



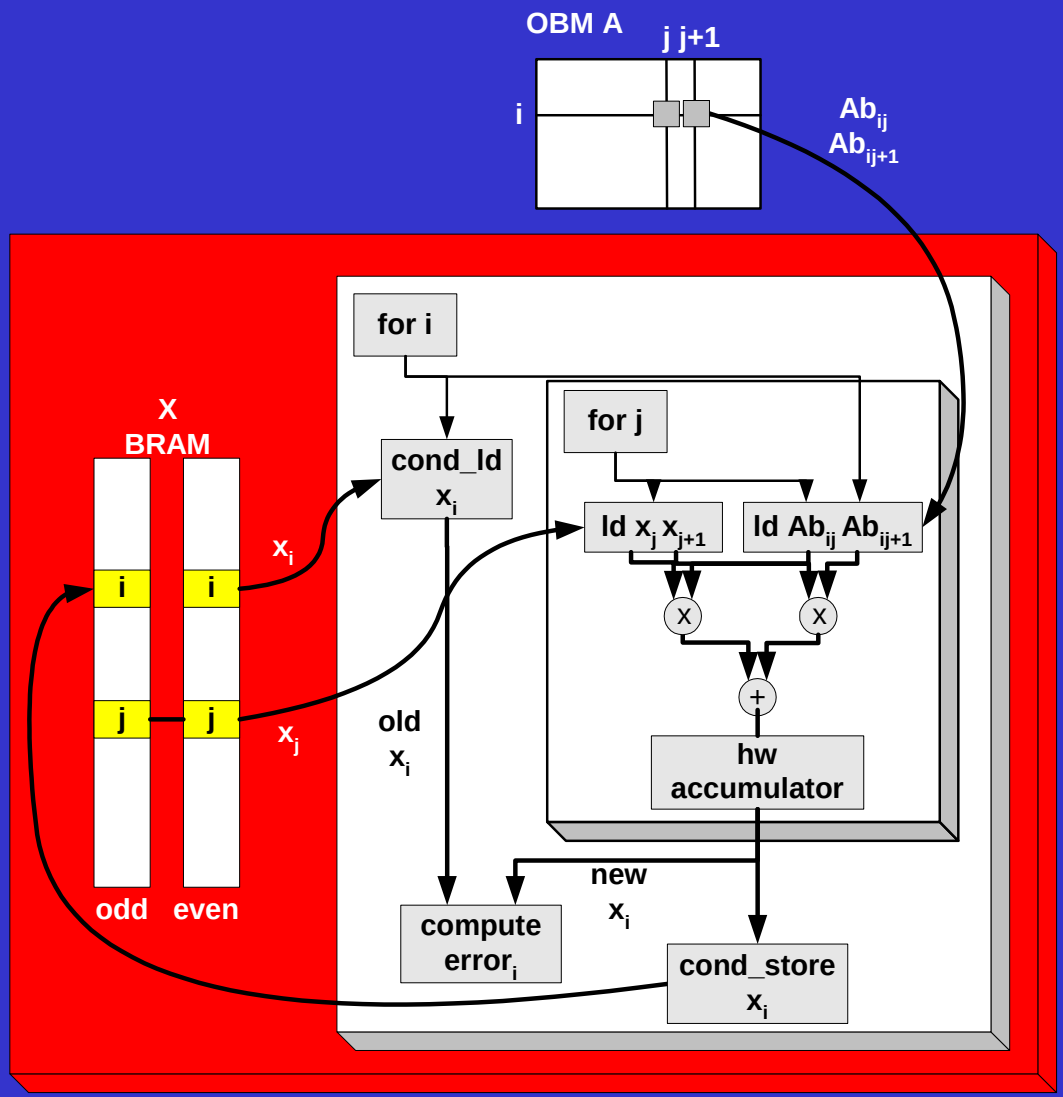
Loop Slowdown caused by Read/Write conflict

Accumulator Macro



Hardware Accumulator macro resolves read / write conflict

Packing the data



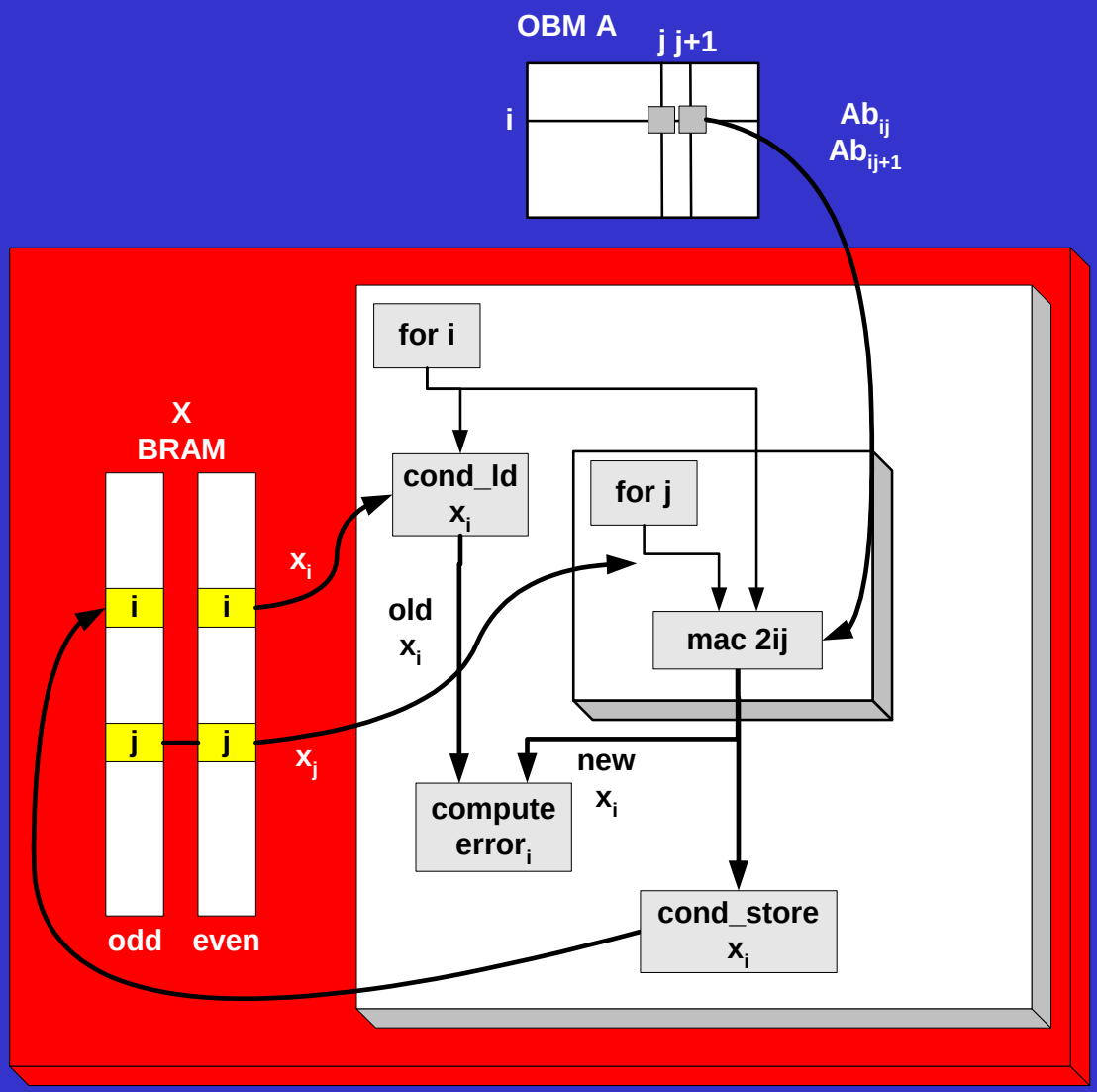
64 bit OBM word can contain two floats

This requires unrolling j loop which now accesses

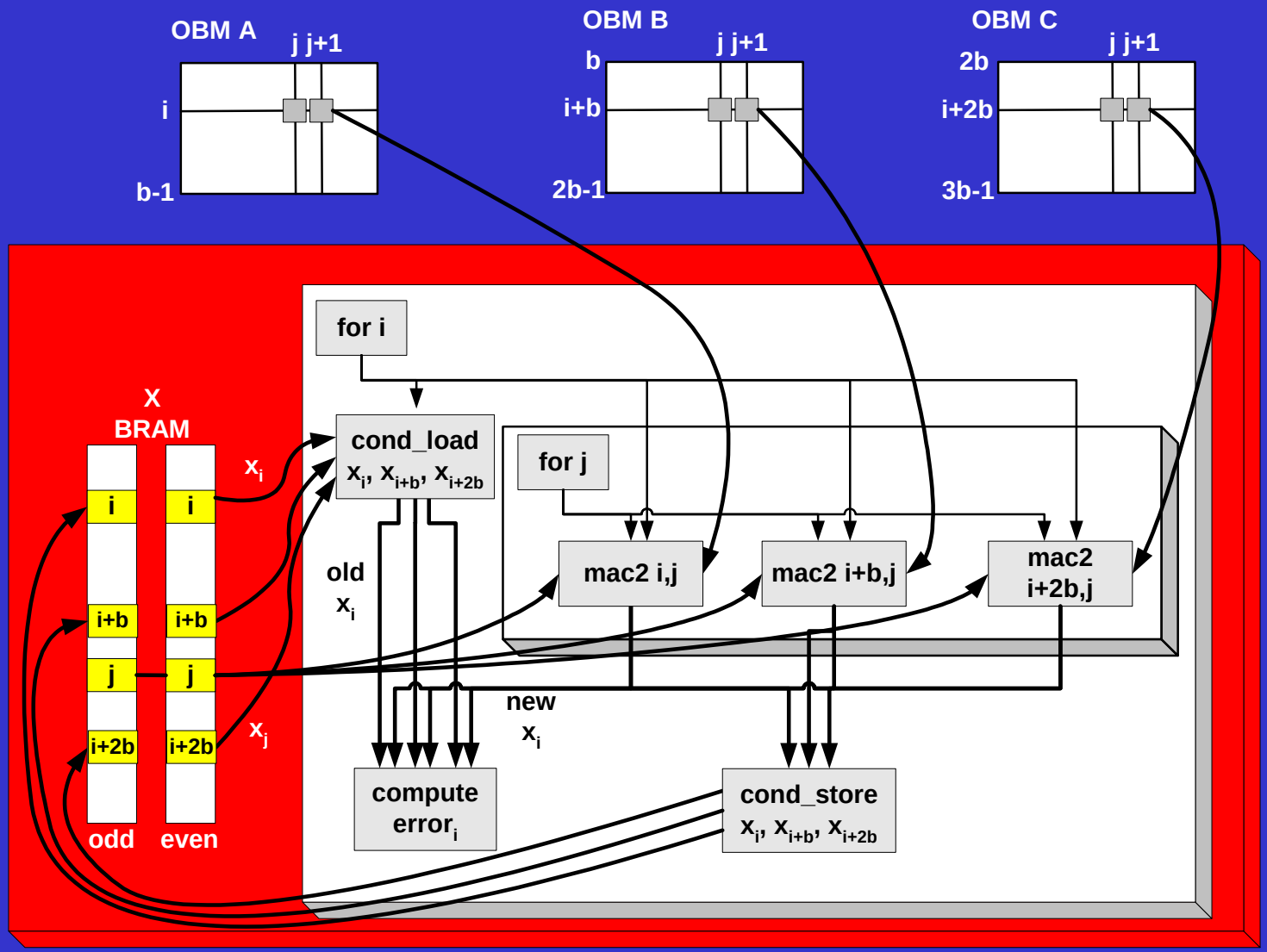
$Ab_{i\,j}$ $Ab_{i\,j+1}$ X_j X_{j+1}

To avoid multiple Block RAM reads, X is stripe partitioned over two Block RAM arrays

Pack Abstracted



Data Partitioned into 3 blocks

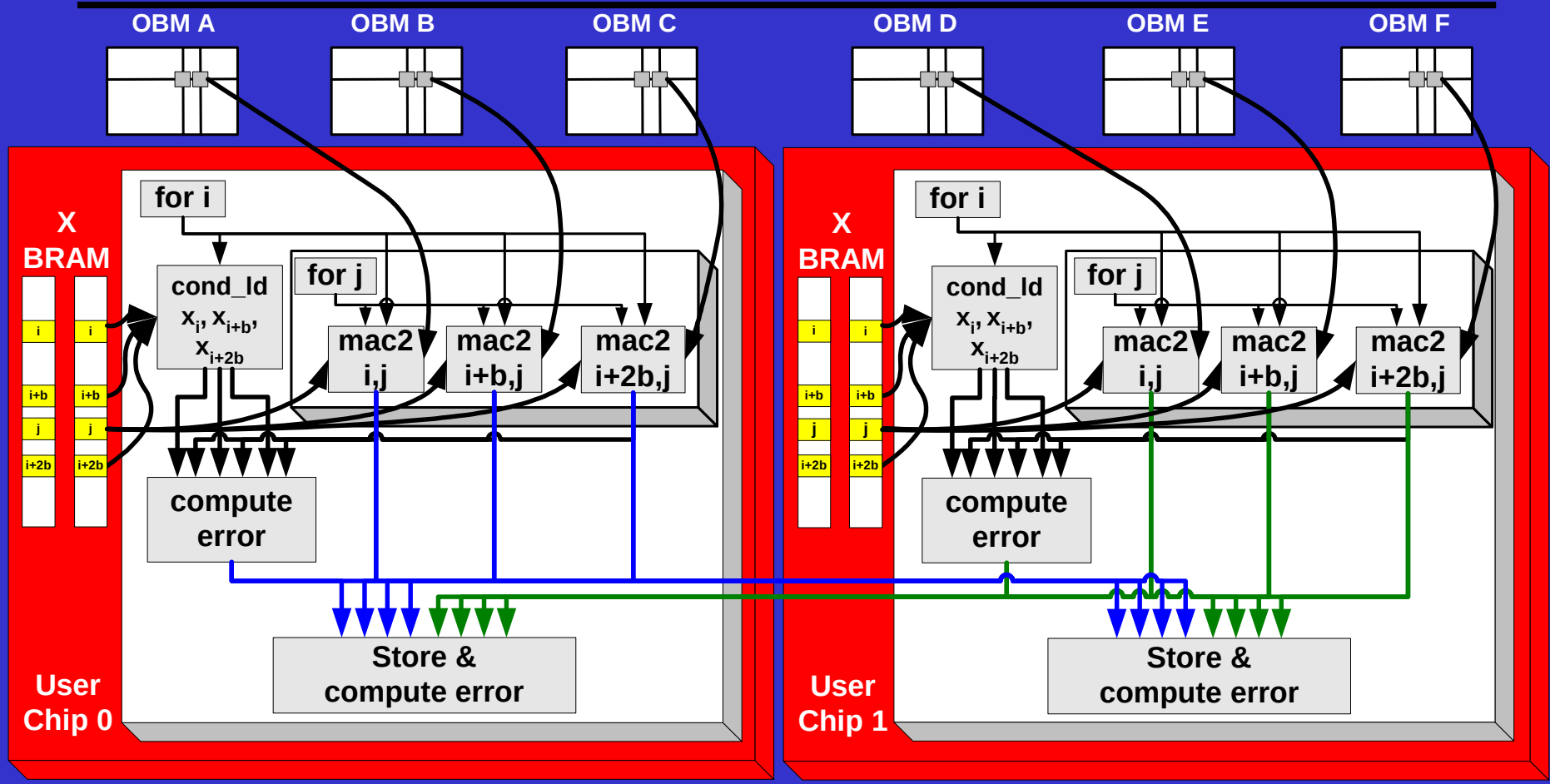


Ab is now row-block distributed (3 blocks in 3 OBMs)

j loop now computes

3 new Xs

Two FPGAs



Ab is row block distributed (6 blocks in 6 OBMs)
The j-loops perform 24 Floating Ops in each clock
FPGA0 and FPGA1 exchange 3 Xs, 1 error

Gauss Seidel Performance

	n=500	n=1000	n=2000
No. Iterations	27	6	7
Pentium IV (2.8 GHz)	0.19 secs 65 MFlops	0.48 secs 26 MFlops	1.90 secs 28 MFlops
MAP	0.013 secs 830 MFlops	0.008 secs 1.23 GFlops	0.031 secs 1.65 GFlops
MAP speedup vs. Pentium	14	57	60

Conclusions

- High Level Algorithmic Language runs on FPGA based HPEC system
  - DEBUG Mode allows most development on workstation
- We can apply standard software design methodologies
  - stepwise refinement
    - currently using macros
    - later using (user directed?) compiler optimizations
- Bandwidth is key to FPGA performance
  - Often, more operations are available in the FPGA fabric than can be supplied by the available off-chip I/O
  - FPGA capability is improving rapidly
- Currently speedups ~50 vs. Pentium IV
- Future: Multiple MAPs
  - More complex, streaming applications